16833 HW 3

Jae Seok Oh (jaeseoko)

1.1

$$r^t = \begin{Bmatrix} r_x^t \\ r_y^t \end{Bmatrix}, \quad r^{t+1} = \begin{Bmatrix} r_x^{t+1} \\ r_y^{t+1} \end{Bmatrix}$$

"odometry"

$$h_o(r^t, r^{t+1}) = \begin{bmatrix} r_x^{t+1} - r_x^t \\ r_y^{t+1} - r_y^t \end{bmatrix}$$

$$H_o(r^t, r^{t+1}) = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

"landmarks"

$$h_l(r^t, l^k) = \begin{bmatrix} l_x^k - r_x^t \\ l_y^k - r_y^t \end{bmatrix}$$

$$H_l(r^t, l^k) = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

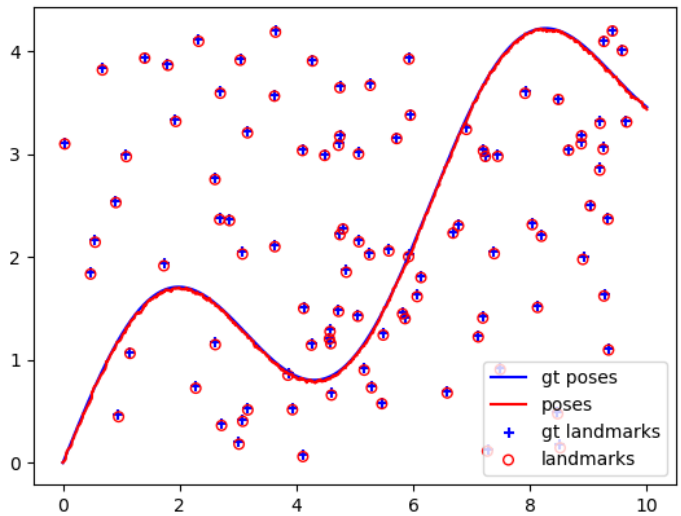
1.2 In linear.py

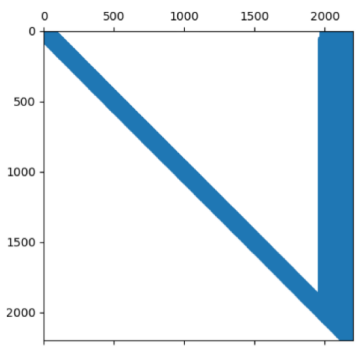
1.3 In solvers.py

1.4.(1,2,3) In solvers.py

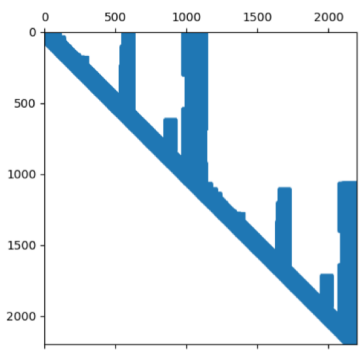
↳ 1.4.2 method & function is "custom_lu"

1.4.4 2D-Linear slam on 2d_linear.npz

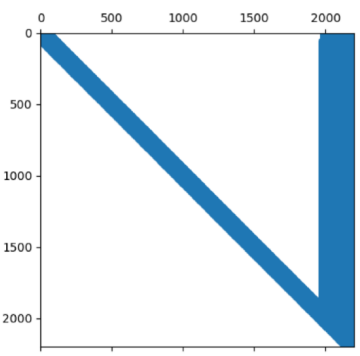




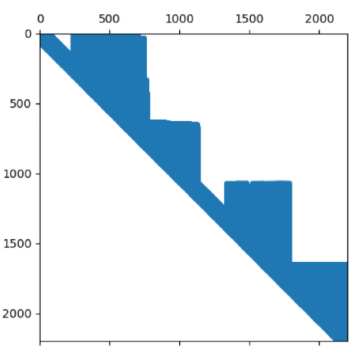
QR



QR colamd



LU



LU colamd

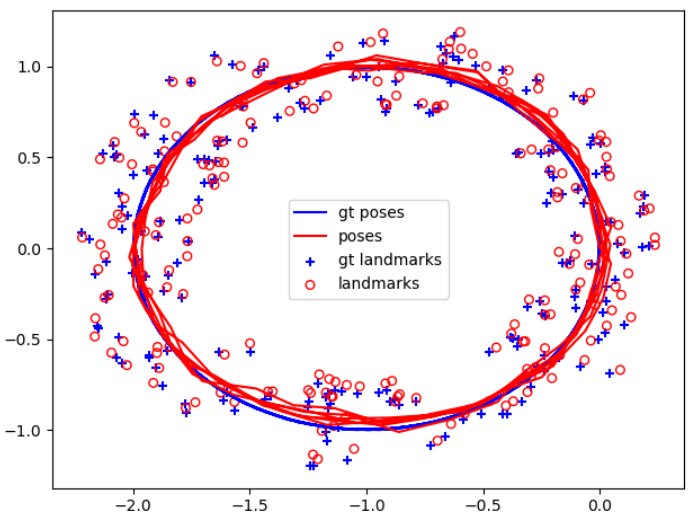
Times	(2d_linear)
PinV	: 3.6467 sec.
QR	: 0.4017 sec.
QR colamd	: 0.2492 sec.
LU	: 0.0303 sec.
LU colamd	: 0.0744 sec.

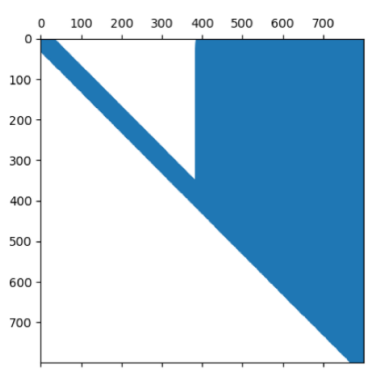
Decomposition in order of efficiency : LU, LU_colamd, QR_colamd, QR, PinV

LU method was the most efficient method for solving the 2d_linear data. LU and LU_colamd performed better than QR and QR_colamd as expected, and pinv was the last since the pseudo inverse of A is very expensive when A is very large like here. We'd expect minimum degree permutation version to be more efficient but here, natural LU performed better. I believe this is due to the fact that natural LU already has a very sparse matrix and the benefit of permuting the rows and columns in advance to LU decomposition to reduce the non-zeros did not outweigh. Also if we look at the U matrix of LU_colamd, it is very dense column-wise which is not good for backward substitution for getting solution x. However, minimum degree permutation did accelerate the computation in the case of QR.

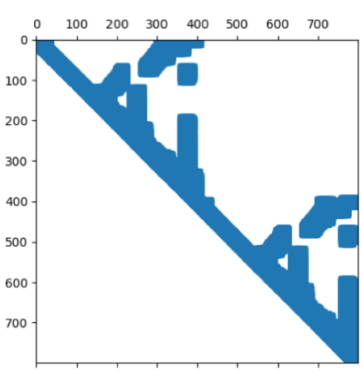
1.4.5

2D linear slam on 2d_linear_loop.npz

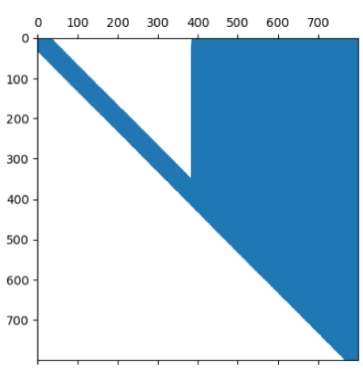




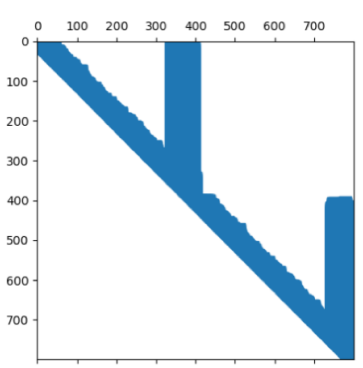
QR



QR colamd



LU



LU (olamd

Times (2d_linear_loop)

PinV	: 0.2599	sec.
QR	: 0.2299	sec.
QR colamd	: 0.0261	sec.
LU	: 0.0243	sec.
LU colamd	: 0.0069	sec.

Decomposition in order of efficiency : LU_colamd, LU, QR_colamd, QR, PinV

For the loop data case, LU_colamd performed the best. Here our A matrix is relatively small, so permutation of the rows and columns would be quick and it does accelerate the computation for both LU and QR. Also, our QR and LU in their natural form are very dense, so permuting the matrix to reduce non-zeros increases the performance. Although pinv is costly, since the A matrix is small, it took relatively similar even though it was still the slowest. Lastly, compared to the linear.npz case, the accuracy on loop.npz is lower by looking at the trajectories and landmark estimations qualitatively because we accumulate drifts as we go around over time without performing any loop closing.

2.1.1

In nonlinear.py

2.1.2

$$h_l(r^t, l^k) = \begin{bmatrix} \operatorname{atan2}(l_y^k - r_y^t,\ l_x^k - r_x^t) \\ \left((l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2\right)^{1/2} \end{bmatrix}$$

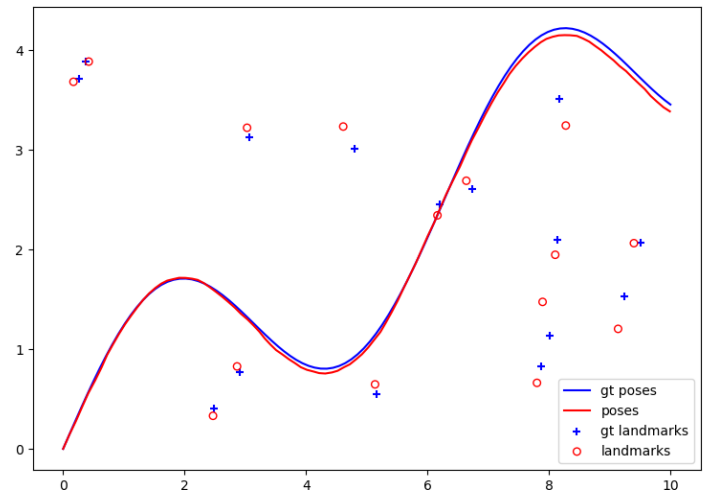
$$H_l(r^t, l^k) =$$

$$\begin{bmatrix} \frac{l_y^k - r_y^t}{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2} & \frac{r_x^t - l_x^k}{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2} & \frac{(r_y^t - l_y^k)}{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2} & \frac{(l_x^k - r_x^t)}{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2} \\ \frac{r_x^t - l_x^k}{\sqrt{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2}} & \frac{r_y^t - l_y^k}{\sqrt{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2}} & \frac{l_x^k - r_x^t}{\sqrt{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2}} & \frac{l_y^k - r_y^t}{\sqrt{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2}} \end{bmatrix}$$

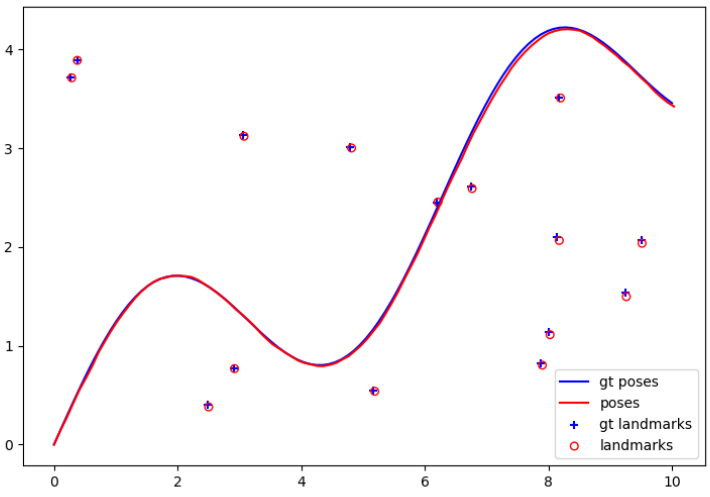
2.2

In nonlinear.py

2.3



Before optimization



After optimization

In the linear case, we implement the least-square optimization as a batch optimization, but in the non-linear case, we cannot do the same. This is because we approximate at a point of linearization and it is only valid very close to that point. We are trying to minimize the original least square problem as in the linear case along with this prediction error from the linear approximation and we do this by iteratively correcting the prediction error.